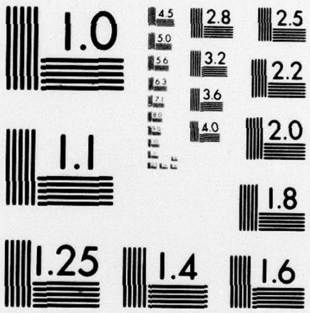
AD-A080 671 CALIFORNIA UNIV BERKELEY F/G 9/2
THE INFLUENCE OF THE COMPILER ON THE COST OF MATHEMATICAL SOFTW--ETC(U)
APR 74 B N PARLETT, Y WANG N00014-69-A-0200-1017

UNCLASSIFIED NL

| OF |
AD
A080671

END
DATE
FILMED
3 - 80
DDC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

The Influence of the Compiler on the Cost of Mathematical Software—in Particular on the Cost of Triangular Factorization

(10) B.N. PARLETT and Y. WANG
University of California, Berkeley

It is suggested that it is almost impossible to compare two sensible implementations of a numerical algorithm in Fortran or in Algol and assert that one of them will lead to a more efficient program in machine language. This is so because the way the computer makes use of registers in the arithmetic unit has a strong influence on running time. Further, the writer of a subprogram to be used by other people is faced with the fact that his program will be used in a variety of compilers and computers.

Key Words and Phrases: compilers, mathematical software, efficiency, portability, triangular factorization
CR Category: 4.0

1. INTRODUCTION

Those who interest themselves in disseminating scientific subprograms suffer in various ways from an inherent contradiction in their work. They wish to produce good programs. Two aspects of the adjective good, namely, recognition of failure and adequate accuracy, do not concern us here. Two other aspects are *machine independence* (portability) and *efficiency* (execution time). Since efficiency depends, to a certain extent, on the computer there exists the *possibility* of conflict between these two goals.

Sometimes there is no difficulty. There are production codes, written entirely in machine language, whose portability is of no interest to the users. At the other extreme are large scale programs of which numerical subprograms form only a tiny part, both in length of code and in execution time. Here the efficiency of the subprogram is of no importance; the adjective good has lost this component in this

A version of this paper was presented at Mathematical Software II, a conference held at Purdue University, West Lafayette, Indiana, May 29–31, 1974.
This work was supported in part by the Office of Naval Research under Contract N00014-69-A-0200-1017.

Authors' addresses: B.N. Parlett, Computer Science Division, University of California, Berkeley CA 94720; Y. Wang, Department of Mathematics, University of California, Berkeley, CA 94720.

particular environment. Often the environment, or scope, of an application will determine the balance between portability and efficiency.

Unfortunately the producer of library subprograms which are to be used frequently by other people does stand (or sit) on the horns of this dilemma. It is no escape to write a different program for every conceivable environment. There exists a large body of users of standardized numerical subprograms who would like to have reliability and swift execution from their (portable) Fortran subroutines. Can they be satisfied? Are the horns of the dilemma really sharp? Can an expert not produce a Fortran or Algol program which is adequately efficient on all relevant computing systems?

The run time efficiency of a Fortran or Algol program depends on the compiler that translates it and on the computer which executes the machine code. One purpose of our study is to show the strength and variability of this dependence.

We took several variants of the fundamental triangular factorization algorithm for a square matrix and tested them on several compilers on two different computers.

It is surprising how sensitive execution time is to tiny, and reasonable, variations in the Fortran program. We hope that others will try similar experiments on other compilers and other machines. Explanation of the anomalies which occur is an enlightening enterprise.

In the last section we draw some conclusions, but we wish to emphasize that we are not criticizing Fortran compilers nor are we advocating a return to assembly language for scientific programming. We do want users to appreciate the complexities involved in writing standard Fortran subroutines for a variety of users.

A program finely tuned for machine X may run on machine Y but be very inefficient. With the advent of fourth generation computing systems, the situation will become even more complicated.

2. TRIANGULAR FACTORIZATION ALGORITHMS

The reader may consult Forsythe and Moler [2] for details concerning the solution of $Ax = b$, given A and b. The major part of the computation is the triangular decomposition $PA \rightarrow LU$, where P is an aptly chosen permutation matrix, L is unit lower triangular with $| l_{ij} | \leq 1$, and U is upper triangular. Then x is obtained by solving $Lc = b$ and $Ux = c$. The factorization may be done either by Gaussian elimination (GE) or by the compact method (Crout). Normally L and U are written over A.

By definition, $(PA)_{ij} = \sum l_{ik}u_{kj}$, $k = 1, \ldots, \min(i, j)$. Thus $u_{ij} = (PA)_{ij} - \sum l_{ik}u_{kj}$, $k = 1, \ldots, i - 1$. Gaussian elimination subtracts one term $l_{ik}u_{kj}$ at a time, whereas the Crout algorithm computes the whole expression as soon as all the terms are known.

GE Algorithm

For $j = 1, 2, \ldots, n - 1$, repeat the following five steps.

1. Find the smallest index p such that $|a_{pj}| = \max |a_{ij}|$ for $i \geq j$.
2. If $a_{pj} = 0$, skip the remaining steps for this value of j.
3. If $p > j$, interchange elements $j, \ldots, n$ of rows j and p.

4. Compute the multipliers: $a_{ij} \leftarrow a_{ij}/a_{jj}$, $i = j + 1, \ldots, n$.
5. $t \leftarrow a_{ij}$, $i = j + 1, \ldots, n$; if $t \neq 0$ then row $i \leftarrow$ row $i - t \times$ row j (only elements $j + 1, \ldots, n$ in each row).

Alternatively, step 5 may be replaced by

5*. $t \leftarrow a_{ji}$, $i = j + 1, \ldots, n$; if $t \neq 0$ then col $i \leftarrow$ col $i - t \times$ col j (only elements $j + 1, \ldots, n$ in each column).

We call the first version elimination by *rows*, and the second, elimination by *columns*.

Crout Algorithm

For $j = 1, 2, \ldots, n - 1$, repeat the following five steps.

1. Find the smallest p such that $|a_{pj}| = \max |a_{ij}|$ for $i \geq j$.
2. If $a_{pj} = 0$, skip the remaining steps for this value of j.
3. If $p > j$, interchange the *whole* of rows j and p.
4. Compute the multipliers: $a_{ij} \leftarrow a_{ij}/a_{jj}$, $i = j + 1, \ldots, n$.
5. Update column $j + 1$: $a_{i,j+1} \leftarrow a_{i,j+1} - \sum a_{ik}\, a_{k,j+1}$, $i = 2, \ldots, n$(the sum runs from $k = 1$ to $m = \min(i - 1, j)$).

Comparison of GE and Crout

Note that steps 1, 2, and 4 are identical in the two processes. However, at step 3 Crout involves more interchanges than does GE; this is mandatory if the inner product in step 5 is to have this simple form. Hence, an advantage for GE. One reward for the extra labor in the Crout reduction is that the matrix L is explicitly available upon termination, whereas in GE the multipliers in each column may not be aligned by rows because of the interchanges. However, for many applications, including the solution of $Ax = b$, L is not needed explicitly. Crout's claim to fame is that it permits the double precision addition of the single precision products in step 5.

If inner products are not so accumulated, then the two algorithms will produce the same results and so are indistinguishable on grounds of accuracy. Moreover, they require the same number of arithmetic operations on the elements of A.

Discussion

Here, it seems to us, is the borderline between numerical analysis and mathematical software. One of us has been teaching for some years that, in the absence of special treatment of inner products, there is nothing to choose between good implementations of either algorithm. Indeed, a major factor in the development of languages such as Fortran and Algol is the idea that *the scientific programmer should not have to know the idiosyncracies of his computer and the vagaries of the compiler in order to write a decent program*. Numerical analysts are encouraged to publish *portable* programs. Indeed, it is nice when a Fortran program written elsewhere actually runs on one's own Fortran system. However, this joy is somewhat dampened if there is a strong possibility that the program will run 50 percent slower than a Fortran implementation which makes the right decisions with respect to the local environment.

3. QUESTIONS CONCERNING IMPLEMENTATION

We now descend to the third level and look at various ways in which the GE and Crout algorithms may be turned into computer programs.

Here are some questions that the programmer must consider. We emphasize again that our aim is not to produce an optimal implementation but to illustrate the effect of certain choices.

Question 1. *Should rows be interchanged physically or by indirect addressing?*

Question 2. *The inner loop in Crout may be implemented in three different ways* (see Section 7 for details). *Are these differences important?* A scheme for computing an a posteriori error bound, which led us to this study, requires that a particular one of the three be used. We had assumed that there would be little to choose between the implementations. Of course the precision in which the sum is accumulated will influence the outcome.

Question 3. *Should* GE *be done by rows or by columns?* In [3], Moler pointed out that on the IBM 360/67 Compiler H, the column implementation is never slower than the row implementation. When page pulls are involved, it is much faster.

Question 4. *How does* GE *compare with Crout?* We made a study of each question and, where the results surprised us, we inspected the assembly language code to find an explanation. We compared just two computers, the CDC 6400 and the IBM 360/50. On the 6400 we tried five Fortran compilers, RUN, RUNW, FUN, FTN, FTN 3.0. On the 360/50 we tried the G and H compilers at maximal efficiency. We used randomly generated full test matrices of orders 25, 50, and 75.

4. THE COMPUTERS AND THE COMPILERS

IBM 360 Model 50

The model 50 is in the middle range of the computers in the 360 family. The execution times relevant to our study are

Operation	*Time* (in microseconds)
fetch or store	5
comparison	5
integer add	5
(short) floating add	10
(long) floating add	10
integer multiply	25
(short) floating multiply	20
(long) floating multiply	50

Short word integers have more significant bits (31) than short floating point numbers (24) and so take longer to multiply.

In a multiprogramming system such as IBM 360/50, it is necessary to distinguish execution time from elapsed time. Execution times were obtained by making two separate runs at nighttime and taking the minimum of the elapsed times between start and finish of the execution. An alternative scheme is to subtract from the central processing unit (CPU) time for a proper run the CPU time for compiling, loading, I/O, and generation of the test matrices. The agreement was good.

CDC 6400

We remark that the IBM long word has 8 bits more than the CDC single precision word.

The CDC 6400 can perform 10 minor cycles in 1 microsecond. The relevant execution times are as follows:

Operation	*Minor Cycles*
fetch	12
store	10
comparison	13
integer add	6
floating add	11
integer multiply	does not exist
floating multiply (high order part)	57
floating multiply (low order part)	57

Remark. There is a hidden catch in the CDC multiply. What is called double precision multiply is simply the lower order part of the product of two single precision numbers. Thus 114 minor cycles are required to obtain the full product of two single precision numbers. What is more, for all the compilers to which we had access, this extra multiplication cannot be invoked without declaring extra double precision variables. For these reasons the cost of accurate accumulation of inner products is over 100 percent more.

Of course complete, single precision CDC arithmetic is adequate for many computations simply because half that precision, supplemented by accumulation of inner products to that precision, would also have been adequate.

Fortran Compilers

The IBM 360 has simpler compilers (which take up less storage) for the smaller models and more sophisticated ones for the larger models. The G compiler was designed for systems with 32,000 short words of storage and does not do explicit optimization. The H compiler goes to the greatest lengths to optimize the object code. This takes more time to accomplish and is designed for machines with at least 64,000 short words of memory. Three different levels of optimization may be selected on the H compiler (0, 1, 2) and we used the highest one (level 2).

On these problems the code produced by the H compiler runs *three times* faster than the output of G.

We have purposely avoided student-oriented compilers which are more concerned with rapid compilation and good diagnostics than with the efficiency of the object code.[1]

Almost 90 percent of the Fortran jobs run at our computer center (at the University of California, Berkeley) use the RUN compiler on the CDC 6400. This was

[1] In Fortran a typical element $A(I, J)$ will be found in location $a_0 + I - 1 + (J - 1) \bullet \mathrm{NDIM}$, where NDIM is the row dimension of A and a_0 is the base address of A. None of the compilers which we used was so naive as to compute this index anew for each reference to an element of A. Instead, the location of the previous reference to A is updated by an appropriate index addition.

issued by CDC in the mid 1960's (as a general purpose, nonoptimizing compiler), and we currently have version 2.3.

It is quite probable that, through ignorance or habit, this compiler is being used for scientific and engineering calculations where the object code produced by a more sophisticated compiler would be more economical for production runs.

Almost 10 percent of the Fortran jobs use RUNW which is a local adaptation (by D. Lindsay) of the cleaned up, well documented version of RUN produced by the University of Washington in 1970. Among other things it uses rounded arithmetic as against RUN's chopped calculations.

In the late 1960's, CDC supplied the FTN compiler which does take the (often significant) time to optimize its object code. On our problems this optimized code cut running time by 25 to 30 percent. Very few people use it. We also used a more recent (1971) version, FTN 3.0, but this is not officially maintained by the computer center, and for us its results were not significantly different from those of FTN, with one surprise.

Last, we used the FUN compiler which, we believe, was an experimental forerunner to FTN. For the most part its results were intermediate between those of RUN and FTN.

5. ACCUMULATION OF INNER PRODUCTS

We would like to make some comments on the accumulation of inner products in extra precision because the importance of this technique is frequently misunderstood. The *given* precision of most scientific computations is either the double or the single precision of the local hardware.

The accumulation of the inner products in step 5 of Crout offers a clever alternative, intermediate between versions 1 and 2. The matrix is retained in the given precision (no doubling of storage) and simply the *additions* in $\sum a_{ik}a_{k,j+1}$ are performed to double accuracy. The assumption here is that the full (doubly precise) result of the given precision multiplication $a_{ik}a_{k,j+1}$ is available at no extra cost. This is not true on CDC equipment, for example, and *cannot be obtained in ANSI Fortran on any machine.*

The reward for the accurate additions is that the error *bounds* on the process are both elegant and almost optimally small. However, we must not be beguiled by the bounds alone. In many, but not all, cases the actual errors in the given precision implementation will already be almost optimal; only the bounds on these errors will be larger. The decision to utilize accurate accumulation of inner products should be made in two stages. Either the extra accuracy is mandatory and the extra cost must be considered part of a feasible implementation (for example, in the computation of the residuals $b - Ax$ in iterative refinement; see [2]), or it is not and the possible benefits must be weighed against the extra cost.

In Wilkinson's early computations (on the pilot ACE and the DEUCE) this extra cost was about 5 percent and the case for accurate accumulation was overwhelming. On the CDC 6000 series the extra cost is at least 100 percent and, it seems to us, the case is almost overwhelmingly against this feature.

With the IBM 360 series we reach the same conclusion for different reasons. The fault lies in the software and not in the hardware, for in ANSI Fortran the product of two given precision numbers is given to that precision, even though available information is discarded in the process. Within the confines of this language one is forced to do an explicit double length multiply on two double length operands whose lower order parts are zero. This increases the cost of accumulation by more than 100 percent. In assembly language the extra cost is negligible because the long add is as fast as the short add and the long sum can be kept in the registers throughout the loop. Here lies the attraction of the Crout version.

For these reasons we decided not to complicate our study by letting the precision of the inner loop be another variable.

6. STUDY 1: HOW SHOULD THE ROW INTERCHANGES BE DONE?

No explicit arithmetic operations are involved in exchanging two elements of an array, but some indexing may be needed in address calculations and that depends on the compiler.

Subsubscripts

This mode of indirect addressing is closest to the spirit of Fortran. Let a one-dimensional array P be used to record the row interchanges. Thus $P(J)$ holds the (original) index of the Jth pivotal row. Initially $P(J) = J$. The (I, J) element of the current permuted matrix is given by $A(IP, J)$ where $IP = P(I)$. In this way there is no cost to interchanging rows, but the reference to *every* array element has been made more complicated. The question is how much?

One-Dimensional Representation

We did choose to implement the *implicit* technique on the Crout reduction, treating A as a one-dimensional array. In other words, we took the indexing out of the hands of the compiler and into our own. This trick violates the spirit of Fortran and is illegal in Algol. We did it to present the implicit version at its best *within the constraints* of Fortran. In machine language one can do better.

Crout (with Implicit Row Interchanges)

Step 5 of the algorithm in Section 2 may be coded as follows.

```
JD = J*NDIM
NMI = N - 1
DO 2 I = 1, NM1
   M = I
   IF (M.GT.J) M = J + 1
   SUM = 0.
   K1 = P(I)............................(p_i, 1)
   IJ1 = JD + K1.......................(p_i, j + 1)
   K2 = JD + P(1)......................(p_1, j + 1)
   DO 1 K = 2, M
   SUM = SUM + A(K1)*A(K2)
   K1 = K1 + NDIM.....................(p_i, k)
1  K2 = JD + P(K)......................(p_k, j + 1)
2  A(IJ1) = A(IJ1) - SUM
```

Discussion/Table I

On the IBM 360/50 the 13 percent advantage of the implicit technique on the G compiler turns into a 40 percent handicap on the H compiler (for 50 × 50 matrices). Note that H cuts the execution time of the explicit version to one-third of its G value, whereas it barely halves the implicit technique. This accounts for the switch.

Tables I and II

		IBM 360/50 execution times		CDC 6400 execution times			
	N	G *compiler*	H *compiler*	RUN	RUNW	FUN	FTN 3.0
			Table I				
Implicit	25	1.225	0.671	0.296	0.295	0.210	0.200
		100%	129.0%	100%	100%	112.3%	117.0%
	50	8.518	4.608	2.032	2.058	1.357	1.345
		100%	141.1%	100%	100%	119.3%	133.6%
	75	27.312	14.52	6.526	6.614	4.289	4.272
		100%	143.2%	100%	100%	124.7%	140.9%
Explicit	25	1.424	0.520	0.353	0.350	0.187	0.171
		116.2%	100%	119.3%	118.6%	100%	100%
	50	9.674	3.265	2.452	2.418	1.137	1.007
		113.6%	100%	120.7%	117.5%	100%	100%
	75	30.892	10.144	7.811	7.713	3.440	3.031
		113.1%	100%	119.7%	116.6%	100%	100%
			Table II				
(0)	25	1.863	0.526	0.243	0.242	0.154	0.152
		133.5%	108.2%	104.7%	104.8%	100%	100%
	50	14.152	3.552	1.528	1.572	0.931	0.936
		149.6%	111.8%	110.4%	115.3%	100%	100%
	75	44.833	11.332	4.709	4.817	2.830	2.836
		148.7%	117.2%	114.7%	119.0%	100%	100%
(1)	25	1.396	0.486	0.232	0.231	0.176	0.165
		100%	100%	100%	100%	114.3%	108.6%
	50	9.461	3.177	1.384	1.363	1.060	0.984
		100%	100%	100%	100%	113.9%	105.1%
	75	30.140	9.668	4.107	4.049	3.199	2.955
		100%	100%	100%	100%	113.0%	104.2%
(2)	25	1.424	0.520	0.353	0.350	0.187	0.171
		102.0%	107.0%	152.2%	151.5%	121.4%	112.5%
	50	9.674	3.265	2.452	2.418	1.137	1.007
		102.3%	102.8%	177.2%	177.4%	122.1%	107.6%
	75	30.892	10.144	7.811	7.713	3.440	3.031
		102.5%	104.9%	190.2%	190.5%	121.6%	106.9%

There is a similar phenomenon in comparing the RUN compilers with the FTN type compilers. Here a 19 percent advantage becomes a 35 to 40 percent disadvantage for the implicit technique.

The fact is that explicit interchanges are not very expensive; $(N - 1)$ of them add only negligibly to the run time. Timings are varied because the number of required interchanges varies. On our random matrices, over $(2/3)N$ interchanges occur.

These ratios would change considerably if the two methods of interchanging were compared in connection with either Gaussian elimination or with Crout using accumulation of inner products.

7. STUDY 2: THE INNER LOOP IN CROUT

We have to implement step 5 of Crout: $a_{i,j+1} \leftarrow a_{i,j+1} - \sum a_{ik}a_{k,j+1}$. Here are three ways of doing it. Method (j) makes use of j temporary variables.

```
(0)     DO 1 K = 1, M
      1 A(I, JP1) = A(I, JP1) - A(I, K)*A(K, JP1)

(1)     SUM = A(I, JP1)
        DO 1, K = 1, M
      1 SUM = SUM - A(I, K)*A(K, JP1)
        A(I, JP1) = SUM

(2)     SUM = A(I, JP1)
        DO 1 K = 1, M
        T = A(I, K)*A(K, JP1)
      1 SUM = SUM - T
        A(I, JP1) = SUM
```

Discussion/Table II

IBM 360/50. With both compilers, (1) is the fastest version but (2) is only 2 percent slower on G and 3 to 6.5 percent slower on H. However, (0) is much worse on G than on H.

CDC 6400. There is no consistent picture here and the variations are large.

RUN and RUNW. (1) is best, with (0) between 4 and 15 percent slower and (2) a whopping 50 to 90 percent slower. This is because the operations required for computing the array locations in the K-loop are done once and for all in (1) but are repeated each time in (2). In other words, the extra variable forced certain indices out of the registers.

FUN, FTN, FTN 3.0. (0) is best because $A(I, J)$ is kept in the registers throughout execution of the whole of the K-loop, and all unnecessary fetches and stores are avoided. Following is a comparison of minor cycles generated by FUN and FTN.

	FUN		FTN	
	(0)	(1)	(0)	(1)
I-loop	411	619	609	801
K-loop	139	161	144	144

The K-loop in (1) also has to fetch and store the temporary variable SUM which is not held in a register throughout the loop.

It is important to note that in this study FTN 3.0 is only marginally better than FTN and for both of them (2) is only from 2 to 2.5 percent slower than (1) and from 2 to 7 percent slower than (0). This is in sharp contrast to the 95 percent cost with the RUN compiler and it is what one would hope to see.

8. STUDY 3: GE BY ROWS VERSUS GE BY COLUMNS

After the row interchanges have been made, it is necessary to execute $a_{ik} \leftarrow a_{ik} - a_{ij} \times a_{jk}$ for i and k from $j + 1$ to n.

```
   GE (rows):
   DO 2 I = JP1, N
   T = A(I, J)
   IF (T.EQ.0.)GO TO 2
   DO 1 K = JP1, N
1  A(I, K) = A(I, K) - T*A(J, K)
2  CONTINUE
```

```
   GE (columns):
   DO 2 K = JP1, N
   T = A(J, K)
   IF (T.EQ.0.) GO TO 2
   DO 1 I = JP1, N
1  A(I, K) = A(I, K) - A(I, J)*T
2  CONTINUE
```

The point here is that Fortran stores matrices by columns and GE (columns) only needs columns j and k in the fast memory at any one time. A clever compiler, however, can produce an equally good code for each case where the whole matrix can be stored in the fast memory.

We draw attention to Moler's discussion [3] of the possible effect of the paging schemes used in time-sharing on the IBM 360/67. He shows that GE (columns) minimizes page pulls in such an environment and is never slower (and sometimes faster) than GE (rows) when a reasonably sophisticated compiler is used. Our results are consistent with this but show that the G compiler cannot be classed as reasonably sophisticated.

Discussion/Table III

It is interesting that the differences are not negligible. On the optimizing compilers (H, FUN, FTN 3.0), GE (rows) is 3 to 10 percent slower than GE (columns).

The RUN compilers cause the usual GE (rows) to be from 20 to 30 percent slower. This is surprising, especially in conjunction with the fact that the non-optimizing G compiler on the IBM 360/50 made GE (columns) 6 or 7 percent slower.

Inspection of the machine code reveals that the slower times are not caused by any flagrant inefficiencies such as recomputing an index but are solely due to the fetching and storing of quantities that could have been held in registers.

For each compiler we compared the best versions of GE and Crout as determined by the previous studies. We omit the timings, which may be found in [4].

9. STUDY 4: CROUT VERSUS GAUSSIAN ELIMINATION

Discussion

There is one anomaly. GE (row) is approximately 10 percent slower than Crout (1) on the G compiler. Apart from this, GE is a comfortable winner, being an amazing 25 percent faster for standard compilers and perhaps 8 percent faster for optimizing compilers.

Table III

		IBM 360/50 execution times		CDC 6400 execution times			
	N	G compiler	H compiler	RUN	RUNW	FUN	FTN 3.0
By rows	25	1.463	0.487	0.212	0.215	0.151	0.148
		100%	109.4%	121.1%	121.5%	107.9%	105.0%
	50	11.070	3.222	1.367	1.347	0.899	0.907
		100%	108.5%	126.5%	124.5%	104.5%	104.6%
	75	33.438	10.208	4.193	4.126	2.732	2.758
		100%	106.8%	128.3%	126.3%	105.0%	103.3%
By columns	25	1.540	0.445	0.175	0.177	0.140	0.141
		105.3%	100%	100%	100%	100%	100%
	50	11.710	2.970	1.081	1.082	0.860	0.867
		105.8%	100%	100%	100%	100%	100%
	75	35.929	9.558	3.267	3.268	2.601	2.671
		107.4%	100%	100%	100%	100%	100%

Recall that for both processes the standard operation count yields $\frac{1}{6}N(N-1)(2N-1)$ multiplications and additions (in the inner loop) plus $\frac{1}{2}N(N-1)$ multiplications by the reciprocal of the pivot for the multipliers. This gives $\frac{1}{3}N^3 + O(N)$ multiplications in all. For contrast we give some actual minor cycle counts on the CDC 6400.

RUN:	Crout (1)	$70N^3 + 450N^2 + O(N)$
	GE (row)	$77N^3 + 58N^2 + O(N)$
FTN 3.0:	Crout (1)	$48N^3 + 328N^2 + O(N)$
	Crout (0)	$48N^3 + 238N^2 + O(N)$

10. CONCLUSIONS

Discussion of our findings with others has pointed up the need for several disclaimers.

We are not implying that programs ought to be written in assembly language code in order to avoid inept use of registers. We are not castigating Fortran compilers for producing imperfect code. We are not blaming anyone nor complaining that this state of affairs is deplorable and should be rectified.

The facts that we have assembled do indicate the nasty problems which beset those who try to disseminate high class numerical programs. Isn't this dissemination one of the aims of the numerical analysis community? Some proliferation of programs is inevitable because of the special forms (such as positive definite matrices, banded matrices) of which advantage can be taken. To check this proliferation our purveyor feels obliged to choose *one* high level (Fortran) implementation for each case. What are his efforts worth if the user comes back to say that the new program seems to be fully as accurate as his 10 year old program but 30 percent slower (or 50 percent slower, or 10 percent slower)? Of course there are two qualities more important than efficiency, namely, recognition of failure and adequate accuracy. Moreover, there are important classes of computation where the scientific calculations are insignificant compared with the other parts. Nevertheless there are a great many installations which use Fortran libraries and where certain subroutines are called repeatedly in other subroutines.

We conclude that the writeups of these subroutines should indicate the difference made by using an optimizing compiler or the standard one. Also helpful would be time estimates not based solely on counts of arithmetic operations. What turns out to be crucial is the way in which the Fortran program influences the compiler's exploitation of the operand registers.

REFERENCES

1. Chartres, B., and Gueder, J. C. Computable error bounds for direct solution of linear equations. *J. ACM 14*, 1 (Jan. 1967), 63–71.
2. Forsythe, G. E., and Moler, C. *Computer Solution of Linear Algebraic Systems.* Prentice-Hall, Englewood Cliffs, N.J., 1967.
3. Moler, C. B. Matrix computations with Fortran and paging. *Comm. ACM 15*, 4 (April 1972), 268–270.
4. Parlett, B. N., and Wang, Y. The influence of the compiler on the cost of triangular factorization. Computer Sci. Tech. Rep. 14, Computer Sci. Div., U. of California, Berkeley, Calif., June 1973.

Received April 1974; revised September 1974